Implicit automata in typed λ-calculi

Nguyễn Lê Thành Dũng (a.k.a. Tito) — nltd@nguyentito.eu
Laboratoire d'informatique de Paris Nord, Villetaneuse, France
joint work with Pierre Pradic (University of Oxford)

Inria / LSV, Deducteam seminar, June 25th, 2020

The idea

Suppose the programs of type T in a programming language $\mathcal{P}$
all compute *languages*, something like $T = \texttt{String} \to \texttt{Bool}$.
(Or *functions* $\texttt{String} \to \texttt{String}$.)

What class of languages? Depends on $\mathcal{P}$ and T.
Many theoretical PLs *not* Turing-complete, especially *typed λ-calculi*
(type systems $\to$ guaranteed termination w/ time bounds)

The idea

Suppose the programs of type T in a programming language $\mathcal{P}$
all compute *languages*, something like $T = \texttt{String} \to \texttt{Bool}$.
(Or *functions* $\texttt{String} \to \texttt{String}$.)

What class of languages? Depends on $\mathcal{P}$ and T.
Many theoretical PLs *not* Turing-complete, especially *typed λ-calculi*
(type systems $\to$ guaranteed termination w/ time bounds)

Implicit complexity: machine-free characterizations of complexity classes
using high-level programming languages

Big project: the same thing for automata instead of complexity

The idea

Suppose the programs of type T in a programming language $\mathcal{P}$
all compute *languages*, something like $T = \mathtt{String} \to \mathtt{Bool}$.
(Or *functions* $\mathtt{String} \to \mathtt{String}$.)

What class of languages? Depends on $\mathcal{P}$ and T.
Many theoretical PLs *not* Turing-complete, especially *typed λ-calculi*
(type systems $\to$ guaranteed termination w/ time bounds)

Implicit complexity: machine-free characterizations of complexity classes
using high-level programming languages

Big project: the same thing for automata instead of complexity

Plan

1. Alternative justification:
   internal motivations from *simply typed λ-calculus*
2. A concrete result: *regular string-to-string functions* in an *affine* λ-calculus
   (+ brief mention of *star-free languages* vs *non-commutative types*)
3. Some abstract nonsense on monoidal closed categories

Simply typed functions on Church numerals (1)

Church encodings of (unary) natural numbers:

- $\mathsf{Nat} = (o \to o) \to o \to o$
- $n \in \mathbb{N} \rightsquigarrow \overline{n} = \lambda f.\ \lambda x.\ f\,(\ldots\ (f\,x)\ldots) : \mathsf{Nat}$ with n times f
- all inhabitants of Nat are equal to some $\overline{n}$ up to $=_{\beta\eta}$

Theorem (Schwichtenberg 1975)

The functions $\mathbb{N} \to \mathbb{N}$ definable by simply-typed λ-terms of type $\mathsf{Nat} \to \mathsf{Nat}$ *are the* extended polynomials (*generated by 0, 1, $+$, $\times$,* `id` *and* `ifzero`).

Simply typed functions on Church numerals (1)

Church encodings of (unary) natural numbers:

- $\mathsf{Nat} = (o \to o) \to o \to o$
- $n \in \mathbb{N} \rightsquigarrow \overline{n} = \lambda f.\, \lambda x.\, f\,(\ldots\ (f\,x)\ldots) : \mathsf{Nat}$ with n times f
- all inhabitants of Nat are equal to some $\overline{n}$ up to $=_{\beta\eta}$

Theorem (Schwichtenberg 1975)

The functions $\mathbb{N} \to \mathbb{N}$ *definable by simply-typed* λ*-terms of type* $\mathsf{Nat} \to \mathsf{Nat}$ *are the* extended polynomials (*generated by 0, 1,* $+$, $\times$, `id` *and* `ifzero`).

Let's add a bit of (meta-level) polymorphism: $t = \mathsf{Nat}[A] \to \mathsf{Nat}$
where $\mathsf{Nat}[A] = \mathsf{Nat}[A/o] = (A \to A) \to A \to A$

Open question

Choose some simple type A and some term $t : \mathsf{Nat}[A] \to \mathsf{Nat}$.
What functions $\mathbb{N} \to \mathbb{N}$ can be defined this way?

Simply typed functions on Church numerals (2)

Take $\mathsf{mult} = \lambda n.\lambda m.\lambda f.\, n\,(m\,f) : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$.
$\mathsf{mult}\ \overline{2} : \mathsf{Nat} \to \mathsf{Nat}$ can be iterated by a $\mathsf{Nat}[\mathsf{Nat}]$...

$$\longrightarrow \mathsf{exp2} = \lambda n.\, n\,(\mathsf{mult}\ \overline{2})\ \overline{1} : \mathsf{Nat}[\mathsf{Nat}] \to \mathsf{Nat}$$

which cannot be iterated!

Simply typed functions on Church numerals (2)

Take $\mathsf{mult} = \lambda n.\lambda m.\lambda f.\, n\,(m\,f) : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$.
$\mathsf{mult}\ \overline{2} : \mathsf{Nat} \to \mathsf{Nat}$ can be iterated by a $\mathsf{Nat}[\mathsf{Nat}]$...

$$\longrightarrow \mathsf{exp2} = \lambda n.\, n\,(\mathsf{mult}\ \overline{2})\ \overline{1} : \mathsf{Nat}[\mathsf{Nat}] \to \mathsf{Nat}$$

which cannot be iterated! Smaller types, still heterogenous:

$$\mathsf{exp2} = \lambda n.\, n\ \overline{2} : \mathsf{Nat}[o \to o] \to \mathsf{Nat}$$

Simply typed functions on Church numerals (2)

Take $\mathsf{mult} = \lambda n.\lambda m.\lambda f.\, n\,(m\,f) : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$.
$\mathsf{mult}\ \overline{2} : \mathsf{Nat} \to \mathsf{Nat}$ can be iterated by a $\mathsf{Nat}[\mathsf{Nat}]$...

$$\longrightarrow \mathsf{exp2} = \lambda n.\, n\,(\mathsf{mult}\ \overline{2})\ \overline{1} : \mathsf{Nat}[\mathsf{Nat}] \to \mathsf{Nat}$$

which cannot be iterated! Smaller types, still heterogenous:

$$\mathsf{exp2} = \lambda n.\, n\ \overline{2} : \mathsf{Nat}[o \to o] \to \mathsf{Nat}$$

Towers of exponentials of *any fixed height* $\mathsf{Nat}[T[A]] \to \mathsf{Nat}[A]$.
This is the fastest possible growth for simply typed λ-terms.

Simply typed functions on Church numerals (2)

Take $\mathsf{mult} = \lambda n.\lambda m.\lambda f.\, n\,(m\,f) : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$.
$\mathsf{mult}\ \overline{2} : \mathsf{Nat} \to \mathsf{Nat}$ can be iterated by a $\mathsf{Nat}[\mathsf{Nat}]$...

$$\longrightarrow \mathsf{exp2} = \lambda n.\, n\,(\mathsf{mult}\ \overline{2})\ \overline{1} : \mathsf{Nat}[\mathsf{Nat}] \to \mathsf{Nat}$$

which cannot be iterated! Smaller types, still heterogenous:

$$\mathsf{exp2} = \lambda n.\, n\ \overline{2} : \mathsf{Nat}[o \to o] \to \mathsf{Nat}$$

Towers of exponentials of *any fixed height* $\mathsf{Nat}[T[A]] \to \mathsf{Nat}[A]$.
This is the fastest possible growth for simply typed λ-terms.

On the other hand:

Theorem (Statman 198X)

Subtraction *cannot be defined by any simply typed* $t : \mathsf{Nat}[A] \to \mathsf{Nat}[B] \to \mathsf{Nat}$.

Simply typed functions on Church numerals (3)

Open question

Choose some simple type A and some term $t : \mathsf{Nat}[A] \to \mathsf{Nat}$.
What functions $\mathbb{N} \to \mathbb{N}$ can be defined this way?

Subtraction cannot be defined; some "easy" 1-variable functions, e.g. $n \mapsto \lfloor \sqrt{n} \rfloor$, are also undefinable.

Does this even admit a *satisfying* answer?

Simply typed functions on Church numerals (3)

Open question

Choose some simple type A and some term $t : \mathsf{Nat}[A] \to \mathsf{Nat}$.
What functions $\mathbb{N} \to \mathbb{N}$ can be defined this way?

Subtraction cannot be defined; some "easy" 1-variable functions, e.g. $n \mapsto \lfloor \sqrt{n} \rfloor$, are also undefinable.

Does this even admit a *satisfying* answer? There is one for *predicates*!

Theorem (Joly 2001)

A subset of $\mathbb{N}^k$ *is decidable by some* $t : \mathsf{Nat}[A_1] \to \cdots \to \mathsf{Nat}[A_n] \to \mathsf{Bool}$ (*where* $\mathsf{Bool} = o \to o \to o$) *if and only if it is* ultimately periodic.

Simply typed functions on Church numerals (3)

Open question

Choose some simple type A and some term $t : \mathsf{Nat}[A] \to \mathsf{Nat}$.
What functions $\mathbb{N} \to \mathbb{N}$ can be defined this way?

Subtraction cannot be defined; some "easy" 1-variable functions, e.g. $n \mapsto \lfloor \sqrt{n} \rfloor$, are also undefinable.

Does this even admit a *satisfying* answer? There is one for *predicates*!

Theorem (Joly 2001)

A subset of $\mathbb{N}^k$ *is decidable by some* $t : \mathsf{Nat}[A_1] \to \cdots \to \mathsf{Nat}[A_n] \to \mathsf{Bool}$ (*where* $\mathsf{Bool} = o \to o \to o$) *if and only if it is* ultimately periodic.

Corollary

If $t : \mathsf{Nat}[A] \to \mathsf{Nat}$ *defines* $f_t : \mathbb{N} \to \mathbb{N}$, *then*
$X \subseteq \mathbb{N}$ *ultimately periodic* $\implies f_t^{-1}(X)$ *ultimately periodic.*

A not quite trivial necessary condition!

Simply typed functions on Church-encoded strings

To gain more insight, let's *generalize*! $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$

Church encodings of *strings* over alphabet $\Sigma = \{a, b\}$:

- $\mathsf{Str}_{\{a,b\}} = (o \to o) \to (o \to o) \to o \to o$
- $abb \in \{a, b\}^* \rightsquigarrow \overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x)) : \mathsf{Str}_\Sigma$

More generally $\mathsf{Str}_\Sigma = (o \to o) \to \ldots |\Sigma| \text{ times} \ldots \to (o \to o) \to o \to o$

Open question

Choose some simple type A and some term $t : \mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$.
What functions $\Gamma^* \to \Sigma^*$ can be defined this way?

Without input type substitutions, an answer is known [Zaionc 1987].

Simply typed functions on Church-encoded strings

To gain more insight, let's *generalize*! $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$

Church encodings of *strings* over alphabet $\Sigma = \{a, b\}$:

- $\mathsf{Str}_{\{a,b\}} = (o \to o) \to (o \to o) \to o \to o$
- $abb \in \{a,b\}^* \rightsquigarrow \overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x)) : \mathsf{Str}_\Sigma$

More generally $\mathsf{Str}_\Sigma = (o \to o) \to \dots |\Sigma| \text{ times} \dots \to (o \to o) \to o \to o$

Open question

Choose some simple type A and some term $t : \mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$.
What functions $\Gamma^* \to \Sigma^*$ can be defined this way?

Without input type substitutions, an answer is known [Zaionc 1987].

An answer for predicates [Hillebrand & Kanellakis 1996]

A subset of Σ^* is decidable by some $t : \mathsf{Str}_\Sigma[A] \to \mathsf{Bool}$
if and only if it is a *regular language*.

Note: unary regular languages $\cong$ ultimately periodic subsets of $\mathbb{N}$

λ-definable functions are regular

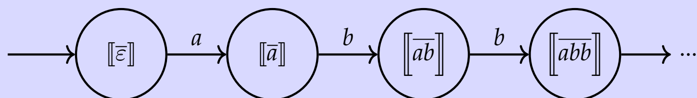
Theorem (Hillebrand & Kanellakis, LICS'96)

For any type A and any simply typed λ-term $t : \mathsf{Str}_\Sigma[A] \to \mathsf{Bool}$, the language $\{w \in \Sigma^* \mid t\,\overline{w} =_\beta \mathtt{true}\}$ is regular.

Proof by semantic evaluation.

Let $[\![-]\!]$ stand for the denotational semantics in the *CCC of finite sets*.

We build an automaton with *finite* set of states $Q = [\![\mathsf{Str}_\Sigma[A]]\!]$ ($\mathrm{Card}(Q)$ depends on A), acceptation as $[\![t]\!](-) = [\![\mathtt{true}]\!]$.



$$t\,\overline{w} =_\beta \mathtt{true} \iff [\![t]\!]([\![\overline{w}]\!]) = [\![\mathtt{true}]\!] \iff w \text{ accepted}$$

(Proof of ($\Leftarrow$): if $\mathrm{Card}([\![o]\!]) \geq 2$ then $[\![\mathtt{true}]\!] \neq [\![\mathtt{false}]\!]$) $\square$

Similar ideas in higher-order model checking, e.g. Grellois & Melliès

Towards implicit automata

Recap: in the simply typed λ-calculus,
$\mathsf{Str}_\Gamma[A] \to \mathsf{Bool}$ = regular languages; $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$ = ???

So what's next?

Towards implicit automata

Recap: in the simply typed λ-calculus,
$\mathsf{Str}_\Gamma[A] \to \mathsf{Bool}$ = regular languages; $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$ = ???

So what's next? Use restricted types to:

- get an easier problem for string-to-string functions

Towards implicit automata

Recap: in the simply typed λ-calculus,
$\mathsf{Str}_\Gamma[A] \to \mathsf{Bool}$ = regular languages; $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$ = ???

So what's next? Use restricted types to:

- get an easier problem for string-to-string functions

- characterize smaller classes of languages
  $\longrightarrow$ *star-free languages* (regular expressions without repetition star, but with complementation)
  using *non-commutative* types (functions must use their arguments in the order that they are given in)
  accepted to ICALP 2020; not covered here, there's a video talk

Towards implicit automata

Recap: in the simply typed λ-calculus,
$\mathsf{Str}_\Gamma[A] \to \mathsf{Bool}$ = regular languages; $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$ = ???

So what's next? Use restricted types to:

- get an easier problem for string-to-string functions
  $\longrightarrow$ **regular functions in an affine λ-calculus**
- characterize smaller classes of languages
  $\longrightarrow$ *star-free languages* (regular expressions without repetition star, but with complementation)
  using *non-commutative* types (functions must use their arguments in the order that they are given in)
  accepted to ICALP 2020; not covered here, there's a video talk

Typing judgments: {non-affine variables} | {affine variables} $\vdash t : A$

- $\lambda^{\to} x.\, t : A \to B$ unrestricted function
- $\lambda^{\circ} x.\, t : A \multimap B$ affine function (at most one x in t)

$$\frac{}{\Gamma, x : A \mid \varnothing \vdash x : A} \qquad \frac{}{\Gamma \mid x : A \vdash x : A}$$

$$\frac{\Gamma \mid \Delta \vdash t : A \to B \qquad \Gamma \mid \varnothing \vdash u : A}{\Gamma \mid \Delta \vdash t\,u : B} \qquad \frac{\Gamma, x : A \mid \Delta \vdash t : B}{\Gamma \mid \Delta \vdash \lambda^{\to} x.\, t : A \to B}$$

$$\frac{\Gamma \mid \Delta \vdash t : A \multimap B \qquad \Gamma \mid \Delta' \vdash u : A}{\Gamma \mid \Delta, \Delta' \vdash t\,u : B} \qquad \frac{\Gamma \mid \Delta, x : A \vdash t : B}{\Gamma \mid \Delta \vdash \lambda^{\circ} x.\, t : A \multimap B}$$

The above = Dual Intuitionistic Linear Logic

$$\frac{\Gamma \mid \Delta \vdash t : A}{\Gamma \mid \Delta' \vdash t : A} \text{ when } \Delta \subseteq \Delta'\text{: } \textit{weakening} \text{ rule}$$

Regular functions

Church encoding with linear/affine types [Girard 1987]:

$$\overline{abb} = \lambda^{\to} f_a.\ \lambda^{\to} f_b.\ \lambda^{\circ} x.\ f_a\ (f_b\ (f_b\ x)) : \mathsf{Str}_{\{a,b\}} = (o \multimap o) \to (o \multimap o) \to o \multimap o$$

Regular functions

Church encoding with linear/affine types [Girard 1987]:

$$\overline{abb} = \lambda^{\to} f_a.\ \lambda^{\to} f_b.\ \lambda^{\circ} x.\ f_a\ (f_b\ (f_b\ x)) : \mathsf{Str}_{\{a,b\}} = (o \multimap o) \to (o \multimap o) \to o \multimap o$$

Theorem (N. & Pradic, unpublished)

$f : \Gamma^* \to \Sigma^*$ *is a* regular function

$$\iff$$

f is defined by some $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ in our affine λ-calculus with A purely affine, *i.e. containing no '$\to$'*

Regular functions

Church encoding with linear/affine types [Girard 1987]:

$$\overline{abb} = \lambda^{\to} f_a.\ \lambda^{\to} f_b.\ \lambda^{\circ} x.\ f_a\ (f_b\ (f_b\ x)) : \mathsf{Str}_{\{a,b\}} = (o \multimap o) \to (o \multimap o) \to o \multimap o$$

Theorem (N. & Pradic, unpublished)

$f : \Gamma^* \to \Sigma^*$ *is a* regular function

$$\Longleftrightarrow$$

f is defined by some $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ in our affine λ-calculus
with A purely affine, *i.e. containing no '$\to$'*

Regular functions are a classical topic, many equivalent definitions...
beware: sequential functions $\neq$ rational functions $\neq$ regular functions

One of them: **copyless** *streaming string transducers* [Alur & Černý 2010]
$\rightsquigarrow$ sounds suspiciously like affine types!

Streaming string transducers

- Finite set of Σ^*-valued *registers* e.g. $R = \{X, Y\}$
- Initial values $R \to \Sigma^*$ e.g. $X_{\text{init}} = Y_{\text{init}} = \varepsilon$
- *Register update function* e.g. $a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases} \quad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$
- "output function" e.g. $\mathtt{out} = XY$

Streaming string transducers

- Finite set of Σ^*-valued *registers* e.g. $R = \{X, Y\}$
- Initial values $R \to \Sigma^*$ e.g. $X_{\text{init}} = Y_{\text{init}} = \varepsilon$
- *Register update function* e.g. $a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases} \quad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$
- "output function" e.g. $\mathtt{out} = XY$

Execution over *abaa*: start with

$$X = \varepsilon \qquad Y = \varepsilon$$

Streaming string transducers

- Finite set of Σ^*-valued *registers* e.g. $R = \{X, Y\}$
- Initial values $R \to \Sigma^*$ e.g. $X_{\text{init}} = Y_{\text{init}} = \varepsilon$
- *Register update function* e.g. $a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases}$ $\quad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$
- "output function" e.g. $\mathsf{out} = XY$

Execution over *abaa*:

$$X = a \qquad Y = a$$

Streaming string transducers

- Finite set of Σ^*-valued *registers* e.g. $R = \{X, Y\}$
- Initial values $R \to \Sigma^*$ e.g. $X_{\text{init}} = Y_{\text{init}} = \varepsilon$
- *Register update function* e.g. $a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases} \quad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$
- "output function" e.g. $\mathtt{out} = XY$

Execution over *abaa*:

$$X = ab \qquad Y = ba$$

Streaming string transducers

- Finite set of Σ^*-valued *registers* e.g. $R = \{X, Y\}$
- Initial values $R \to \Sigma^*$ e.g. $X_{\text{init}} = Y_{\text{init}} = \varepsilon$
- *Register update function* e.g. $a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases} \quad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$
- "output function" e.g. $\mathtt{out} = XY$

Execution over *abaa*:

$$X = aba \qquad Y = aba$$

Streaming string transducers

- Finite set of Σ^*-valued *registers* e.g. $R = \{X, Y\}$
- Initial values $R \to \Sigma^*$ e.g. $X_{\text{init}} = Y_{\text{init}} = \varepsilon$
- *Register update function* e.g. $a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases} \quad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$
- "output function" e.g. $\mathtt{out} = XY$

Execution over *abaa*:

$$X = abaa \qquad Y = aaba$$

Streaming string transducers

- Finite set of Σ^*-valued *registers* e.g. $R = \{X, Y\}$
- Initial values $R \to \Sigma^*$ e.g. $X_{\text{init}} = Y_{\text{init}} = \varepsilon$
- *Register update function* e.g. $a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases} \quad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$
- "output function" e.g. $\mathtt{out} = XY$

Execution over *abaa*: $f(abaa) = abaaaaba$

$$X = abaa \qquad Y = aaba$$

- Finite set of Σ^*-valued *registers* e.g. $R = \{X, Y\}$
- Initial values $R \to \Sigma^*$ e.g. $X_{\text{init}} = Y_{\text{init}} = \varepsilon$
- *Register update function* e.g. $\quad a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases} \quad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$
- "output function" e.g. $\texttt{out} = XY$

Execution over *abaa*: $f(abaa) = abaaaaba$, $f : w \mapsto w \cdot \texttt{reverse}(w)$

$$X = abaa \qquad Y = aaba$$

Streaming string transducers

- Finite set of Σ^*-valued *registers* e.g. $R = \{X, Y\}$
- Initial values $R \to \Sigma^*$ e.g. $X_{\text{init}} = Y_{\text{init}} = \varepsilon$
- *Register update function* e.g. $a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases} \quad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$
- "output function" e.g. $\texttt{out} = XY$

Execution over *abaa*: $f(abaa) = abaaaaba$, $f : w \mapsto w \cdot \texttt{reverse}(w)$

$$X = abaa \qquad Y = aaba$$

SSTs can also have *states*: their memory is $Q \times (\Sigma^*)^R$ (with $|Q| < \infty$)

- Finite set of Σ^*-valued *registers* e.g. $R = \{X, Y\}$
- Initial values $R \to \Sigma^*$ e.g. $X_{\text{init}} = Y_{\text{init}} = \varepsilon$
- *Register update function* e.g. $a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases} \quad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$
- "output function" e.g. $\mathtt{out} = XY$

Execution over *abaa*: $f(abaa) = abaaaaba$, $f : w \mapsto w \cdot \mathtt{reverse}(w)$

$$X = abaa \qquad Y = aaba$$

SSTs can also have *states*: their memory is $Q \times (\Sigma^*)^R$ (with $|Q| < \infty$)

Copyless SST: each register appears *at most once* on RHS of :=

(for each fixed input letter, at most once among all the associated :=)

consider the memory as $M = Q \otimes \Sigma^* \otimes \ldots \otimes \Sigma^*$, a transition is $M \multimap M$

$(Q \cong 1 \oplus \ldots \oplus 1, \mathsf{concat} : \Sigma^* \otimes \Sigma^* \multimap \Sigma^*)$

Additive connectives for streaming tree transducers

Copyless streaming *tree* transducers $\subset$ regular *tree* functions;
conjectured to be a *strict inclusion*.
To recover an equality: ad-hoc relaxation called “single use restriction”.

Principled explanation via linear logic:
just allow the *additive conjunction* in the internal memory!

$$\text{e.g.} \qquad M = Q \otimes \Sigma^* \otimes (\Sigma^* \mathbin{\&} \Sigma^*)$$

(For binary trees $\to$ strings, transitions are $M \multimap M \multimap M$)

Additive connectives for streaming tree transducers

Copyless streaming *tree* transducers $\subset$ regular *tree* functions;
conjectured to be a *strict inclusion*.
To recover an equality: ad-hoc relaxation called "single use restriction".

Principled explanation via linear logic:
just allow the *additive conjunction* in the internal memory!

$$\text{e.g.} \qquad M = Q \otimes \Sigma^* \otimes (\Sigma^* \mathbin{\&} \Sigma^*)$$

(For binary trees $\to$ strings, transitions are $M \multimap M \multimap M$)

Theorem

- $\mathsf{Tree}_\Gamma[A] \multimap \mathsf{Bool}$ *cannot define all regular tree languages*
  (*proved using tree-walking automata and geometry of interaction*)
- *But if you add* additive connectives, *it works,*
  and $\mathsf{Tree}_\Gamma[A] \multimap \mathsf{Tree}_\Sigma$ *gives you all regular tree functions!*

Additive connectives for streaming tree transducers

Copyless streaming *tree* transducers $\subset$ regular *tree* functions;
conjectured to be a *strict inclusion*.
To recover an equality: ad-hoc relaxation called "single use restriction".

Principled explanation via linear logic:
just allow the *additive conjunction* in the internal memory!

$$\text{e.g.} \qquad M = Q \otimes \Sigma^* \otimes (\Sigma^* \mathbin{\&} \Sigma^*)$$

(For binary trees $\to$ strings, transitions are $M \multimap M \multimap M$)

Theorem

- $\mathsf{Tree}_\Gamma[A] \multimap \mathsf{Bool}$ *cannot define all regular tree languages* (*proved using tree-walking automata and geometry of interaction*)
- *But if you add* additive connectives, *it works,* *and* $\mathsf{Tree}_\Gamma[A] \multimap \mathsf{Tree}_\Sigma$ *gives you all regular tree functions!*

Next slides: *category-theoretic* justification
memories/transitions should be objects/morphisms in a nice category
$\longrightarrow$ will help us prove our main theorems (for both strings and trees)

Additive connectives for streaming tree transducers

Copyless streaming *tree* transducers $\subset$ regular *tree* functions;
conjectured to be a *strict inclusion*.
To recover an equality: ad-hoc relaxation called "single use restriction".

Principled explanation via linear logic:
just allow the *additive conjunction* in the internal memory!

$$\text{e.g.} \qquad M = Q \otimes \Sigma^* \otimes (\Sigma^* \mathbin{\&} \Sigma^*) = \bigoplus_{q \in Q} \Sigma^* \otimes (\Sigma^* \mathbin{\&} \Sigma^*)$$

(For binary trees $\to$ strings, transitions are $M \multimap M \multimap M$)

Theorem

- $\mathsf{Tree}_\Gamma[A] \multimap \mathsf{Bool}$ *cannot define all regular tree languages*
  (*proved using tree-walking automata and geometry of interaction*)
- *But if you add* additive connectives, *it works,*
  and $\mathsf{Tree}_\Gamma[A] \multimap \mathsf{Tree}_\Sigma$ *gives you all regular tree functions!*

Next slides: *category-theoretic* justification
memories/transitions should be objects/morphisms in a nice category
$\longrightarrow$ will help us prove our main theorems (for both strings and trees)

Categorical automata theory meets denotational semantics (1)

A framework for "single-pass" automata [Colcombet & Petrişan 2017]

- internal memory = object of a *category*
- transitions = morphisms (and [letter $\mapsto$ transition] = functor)

- DFA = automata over the category of finite sets
- Copyless SSTs $\approx$ start from a category $\mathcal{R}$ of copyless register updates
  + add states by *free finite coproduct completion* $(-)_\oplus$

Categorical automata theory meets denotational semantics (1)

A framework for "single-pass" automata [Colcombet & Petrişan 2017]

- internal memory = object of a *category*
- transitions = morphisms (and [letter $\mapsto$ transition] = functor)

- DFA = automata over the category of finite sets
- Copyless SSTs $\approx$ start from a category $\mathcal{R}$ of copyless register updates
  + add states by *free finite coproduct completion* $(-)_{\oplus}$
- Transductions definable in affine λ-calculus can be turned into automata over a category $\mathcal{L}$ of purely affine λ-terms (w/ const $f_c : o \multimap o$ for $c \in \Sigma$)

Categorical automata theory meets denotational semantics (1)

A framework for "single-pass" automata [Colcombet & Petrişan 2017]

- internal memory = object of a *category*
- transitions = morphisms (and [letter $\mapsto$ transition] = functor)

- DFA = automata over the category of finite sets
- Copyless SSTs $\approx$ start from a category $\mathcal{R}$ of copyless register updates
  + add states by *free finite coproduct completion* $(-)_\oplus$
- Transductions definable in affine λ-calculus can be turned into automata over a category $\mathcal{L}$ of purely affine λ-terms (w/ const $f_c : o \multimap o$ for $c \in \Sigma$)

Proof strategy for affinely λ-definable $\implies$ regular function

Define a *functor* $\mathcal{L} \to \mathcal{R}_\oplus$ preserving enough structure

Useful fact: there is a canonical functor from $\mathcal{L}$ to any *affine symmetric monoidal closed category* (i.e. categorical semantics of the purely affine λ-calculus)

Unfortunately $R_\oplus$ is not monoidal closed...

Categorical automata theory meets denotational semantics (2)

- $\mathcal{L}$: category of purely affine λ-terms (w/ const $f_c : o \multimap o$ for $c \in \Sigma$)
- $\mathcal{R}$: category of finite sets of registers and copyless assignments
- $\mathcal{R}_\oplus$: free finite coproduct completion of the latter (add states)

Inspired by streaming *tree* transducers,
we consider the free finite *product* completion: $\mathcal{C}_\& = ((\mathcal{C}^{\mathrm{op}})_\oplus)^{\mathrm{op}}$

Categorical automata theory meets denotational semantics (2)

- $\mathcal{L}$: category of purely affine λ-terms (w/ const $f_c : o \multimap o$ for $c \in \Sigma$)
- $\mathcal{R}$: category of finite sets of registers and copyless assignments
- $\mathcal{R}_\oplus$: free finite coproduct completion of the latter (add states)

Inspired by streaming *tree* transducers,
we consider the free finite *product* completion: $\mathcal{C}_\& = ((\mathcal{C}^{\mathrm{op}})_\oplus)^{\mathrm{op}}$

$$M = \bigoplus_{u \in U} \ \mathop{\&}_{x \in X(u)} \text{(some object of } \mathcal{R} \text{ depending on } u, x)$$

with $A \otimes (B \mathbin{\&} C) \equiv (A \otimes B) \mathbin{\&} (A \otimes C)$, $A \mathbin{\&} (B \oplus C) \equiv (A \mathbin{\&} B) \oplus (A \mathbin{\&} C)$
non-standard distributivities $\implies$ prenex forms

similar to de Paiva's *Dialectica* categories **DC**, think $\exists u.\, \forall x.\, \varphi(u, x)$
(also related to *lenses* / bidirectional programming)

Categorical automata theory meets denotational semantics (2)

- $\mathcal{L}$: category of purely affine λ-terms (w/ const $f_c : o \multimap o$ for $c \in \Sigma$)
- $\mathcal{R}$: category of finite sets of registers and copyless assignments
- $\mathcal{R}_\oplus$: free finite coproduct completion of the latter (add states)

Inspired by streaming *tree* transducers,
we consider the free finite *product* completion: $\mathcal{C}_\& = ((\mathcal{C}^{\mathrm{op}})_\oplus)^{\mathrm{op}}$

$$M = \bigoplus_{u \in U} \mathop{\&}_{x \in X(u)} (\text{some object of } \mathcal{R} \text{ depending on } u, x)$$

with $A \otimes (B \& C) \equiv (A \otimes B) \& (A \otimes C)$, $A \& (B \oplus C) \equiv (A \& B) \oplus (A \& C)$
non-standard distributivities $\implies$ prenex forms

similar to de Paiva's *Dialectica* categories **DC**, think $\exists u.\, \forall x.\, \varphi(u, x)$
(also related to *lenses* / bidirectional programming)

Theorem (Dialectica + combinatorics on strings)

$(\mathcal{R}_\&)_\oplus$ *is symmetric monoidal closed (and almost affine).*

$\implies$ as we wanted, $(\mathcal{R}_\&)_\oplus$-automata are at least as powerful as $\mathcal{L}$-automata!

A monoidal closure theorem

Lemma (is this folklore?)

If $\mathcal{R}$ is symmetric monoidal and $(\mathcal{R}_{\&})_{\oplus}$ has the internal homs $A \multimap B$ for all $A, B \in \mathcal{R}$, then $(\mathcal{R}_{\&})_{\oplus}$ is symmetric monoidal closed.

$$\left(\bigoplus_{u \in U} \bigwith_{x \in X_u} A_x\right) \multimap \left(\bigoplus_{v \in V} \bigwith_{y \in Y_v} B_y\right) = \bigwith_{u \in U} \bigoplus_{v \in V} \bigwith_{y \in Y_v} \bigoplus_{x \in X_u} A_x \multimap B_y$$

A monoidal closure theorem

Lemma (is this folklore?)

If $\mathcal{R}$ is symmetric monoidal and $(\mathcal{R}_{\&})_{\oplus}$ has the internal homs $A \multimap B$ for all $A, B \in \mathcal{R}$, then $(\mathcal{R}_{\&})_{\oplus}$ is symmetric monoidal closed.

$$\left(\bigoplus_{u \in U} \bigwith_{x \in X_u} A_x\right) \multimap \left(\bigoplus_{v \in V} \bigwith_{y \in Y_v} B_y\right) = \bigwith_{u \in U} \bigoplus_{v \in V} \bigwith_{y \in Y_v} \bigoplus_{x \in X_u} A_x \multimap B_y$$

Lemma

$\mathcal{R}_{\oplus}$ has the internal homs $A \multimap B$ for all $A, B \in \mathcal{R}$.

The construction appears in the original SST paper [Alur & Černý 2010] without the categorical vocabulary.

$$\begin{cases} X := abXcY \\ Y := ba \end{cases} \rightsquigarrow \quad \text{shape} \begin{cases} X := Z_1 X Z_2 Y \\ Y := Z_3 \end{cases} \quad + \quad \text{parameters } Z_1 = ab, \ldots$$

copyless SST $\implies$ finitely many shapes: use as states; registers for params

Miscellaneous comments

Theorem (Dialectica + combinatorics on strings)

$(\mathcal{R}_{\&})_{\oplus}$ *is symmetric monoidal closed (and almost affine).*

$\Longrightarrow$ as we wanted, $(\mathcal{R}_{\&})_{\oplus}$-automata are at least as powerful as $\mathcal{L}$-automata!

- *semantic evaluation* technique, as for STLC $\subseteq$ regular languages
- justification of definition of streaming tree transducers

Miscellaneous comments

Theorem (Dialectica + combinatorics on strings)

$(\mathcal{R}_{\&})_{\oplus}$ *is symmetric monoidal closed (and almost affine).*

$\Longrightarrow$ as we wanted, $(\mathcal{R}_{\&})_{\oplus}$-automata are at least as powerful as $\mathcal{L}$-automata!

- *semantic evaluation* technique, as for STLC $\subseteq$ regular languages
- justification of definition of streaming tree transducers
- monoidal closure = function space construction available
  explains why regular functions are *closed under composition*

Miscellaneous comments

Theorem (Dialectica + combinatorics on strings)

$(\mathcal{R}_{\&})_{\oplus}$ *is symmetric monoidal closed (and almost affine).*

$\Longrightarrow$ as we wanted, $(\mathcal{R}_{\&})_{\oplus}$-automata are at least as powerful as $\mathcal{L}$-automata!

- *semantic evaluation* technique, as for STLC $\subseteq$ regular languages
- justification of definition of streaming tree transducers
- monoidal closure = function space construction available
  explains why regular functions are *closed under composition*
- elimination of '&' in SST $\approx$ determinization

Miscellaneous comments

Theorem (Dialectica + combinatorics on strings)

$(\mathcal{R}_{\&})_{\oplus}$ *is symmetric monoidal closed (and almost affine).*

$\Longrightarrow$ as we wanted, $(\mathcal{R}_{\&})_{\oplus}$-automata are at least as powerful as $\mathcal{L}$-automata!

- *semantic evaluation* technique, as for STLC $\subseteq$ regular languages
- justification of definition of streaming tree transducers
- monoidal closure = function space construction available
  explains why regular functions are *closed under composition*
- elimination of '&' in SST $\approx$ determinization

Why bother with affine λ-calculus *without additives*?

- necessary for star-free language stuff
- geometry of interaction can be used
  $\longrightarrow$ show that every regular function can be computed by
  a two-way finite state transducer
- however, it's non-trivial to encode stateful SSTs without additives...
  (solution: Krohn–Rhodes decomposition of monoids)

Conclusion

$\mathsf{Str}_\Sigma[A] \multimap \mathsf{Bool}$ with A affine (adapted as needed):

λ-calculus	languages	status
simply typed	regular	✓[Hillebrand & Kanellakis 1996]
linear or affine	regular	✓[N. & Pradic 2020]
non-commutative linear or affine	star-free	✓[N. & Pradic 2020]

$\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ with A affine (adapted as needed):

λ-calculus	transducers	status
linear (without additives)	nothing interesting (?)	✓
affine	regular functions	✓(coming soon)
non-commutative affine	first-order regular fn.	✓?
linear/affine with additives	regular functions	✓(coming soon)
parsimonious	polyregular	??
simply typed	variant of CPDA???	???

- Church encodings lead to connections with automata
- Additive connectives are important for trees
- Application of categorical semantics (Dialectica, GoI)